Analysis of algorithms to solve the TSP

Gautham Gururajan[*] Aranya Banerjee Vishal Hariharan

Abstract

The Traveling Salesman Problem (TSP) in one of the fundamental problems in the field of Computer Science and is studied extensively to evaluate performance of new algorithms developed. In this work, 4 different algorithms - Branch and Bound, Approximation algorithm using Minimum Spanning tree (MST) heuristic, Genetic Algorithm, and simulated annealing are implemented on wide range of graph instances. To compare the performance of the algorithms implemented, metrics such as run-time and relative solution qualities are compared across the different algorithms through QRTDs, SQDs, and box plots. Relative strengths and weakness are discussed and best algorithm based on emperical performance is reported.

CCS Concepts: **• Computer systems organization → Embedded systems**; *Redundancy*; Robotics; **• Networks →** Network reliability.

Keywords: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Gautham Gururajan, Aranya Banerjee, and Vishal Hariharan. 2018. Analysis of algorithms to solve the TSP. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

The traveling salesman problem, the problem of finding the shortest hamiltonian cycle in a graph, offers a broad range of applications and algorithm benchmarking opportunities. The simplest application lies in the task of visiting locations using a tour of minimum distance. This work applies branch and bound, minimum spanning tree heuristic, simulated annealing, and genetic algorithms to this task, and the results of applying these algorithms over points of interest in various cities indeed meet theoretical expectations. The Branch and Bound is seen to give exact solutions eventually but at the cost of time complexity, the MST approximation algorithm gives us extremely good results and peaks utility with it's ease of implementation. Our local search heuristics show interesting adaptations of natural sciences in the area of core theoretical computation with the Simulated Annealing procedure being the faster to converge of the two. The Genetic algorithm is also seen to have it's own advantages in being able to reach the neighborhood of the solution fairly quickly, but cannot seem to converge to the exact solution in short amounts of time. The report goes on to explain each algorithm in detail with respect to the tracvv

[*]All authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/1122445.1122456

2 Problem Definition

This work addresses the Metric TSP problem in two dimensions, given in definition 2.1 and defines the cost function c using Euclidean distance.

Definition 2.1. 2D Metric TSP: Given N points in the $x - y$ plane with metric cost function c defined for every pair of points, find the shortest simple cycle that visits all N points. A metric cost function is one that is symmetric and that satisfies the triangle inequality.

3 Related Work

3.1 Branch and Bound

First formulated in [7], the branch and bound paradigm was proposed as a framework characterized by efficiently considering the space of candidate solutions to optimization problems. The idea is to consider each candidate solution according to how promising its objective function is before exploring the candidate solutions that stem from it. A branch and bound algorithm using minimum spanning trees to form a lower bound was first presented in [4]. Our implementation's details are presented in section 4.1.

3.2 Approximation

Approximation algorithms use heuristics to produce solutions within some guaranteed quality relative to the optimal solution. Construction heuristics for approximation of the TSP problem include random, closest, and farthest insertion of vertices into existing tours as well as selection of the nearest unvisited vertex to add to the tour [3]. This work focuses on the use of minimum spanning trees, first presented to obtain a 2-approximation in [10]. The main idea is to construct a shortcutted Eulerian tour of the doubled edges in of the graph's minimum spanning tree. The best polynomial time approximation algorithm well-known in literature is the Christofides–Serdyukov algorithm, which uses uses additional optimization in the conversion from the minimum spanning tree to the Eulerian tour by finding a minimum-weighted perfect matching for the odd-degree nodes in the minimum-spanning tree to obtain 3/2 approximation [9]. Our implementation's details are presented in 4.2.

3.3 Simulated Annealing

This report talks about two local search problems. The first of these is Simulated Annealing, which has it's name come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. These concepts are fundamental to the creation of the algorithm. It was independently introduced by many authors, the first of which was M.Pincus in 1970 [8]. Details are presented in 4.3.

3.4 Genetic Algorithm

This is another type of local search algorithm that holds it's strength from evolutionary principles. It was developed by John Holland and his collaborators in the 1960s and 1970s (Holland, 1975; De Jong, 1975 [5]), and is a model or abstraction of biological evolution based on Charles Darwin's theory of natural selection. Crossover and recombination, mutation, and selection are genetic operators that are the backbone of this algorithm. In previous work by Fogel et. al [2] the TSP was discussed, with results not far from intuition - details are presented in 4.6.

4 Algorithms

In the following subsections, we discuss our choice of algorithms in detail and also attach their corresponding pseudocodes.

4.1 Branch and Bound

4.1.1 Algorithm Description. This algorithm follows the branch and bound paradigm of choosing the best candidate solution based on its lower bound cost and either expanding it to new candidate solutions, rejecting it as a complete solution that is a dead end, or accepting it as a new complete solution that is a new best solution. In this algorithm, given a candidate tour over some subset of vertices, the lower bound is calculated from the cost of the candidate tour plus the cost of the minimum spanning tree of the unvisited vertices plus the cost of the cheapest edges from each end of the candidate tour to the univisted vertices. The pseudocode is given in Algorithm 1.

4.1.2 Approximation Guarantee. If allowed to run to termination, the branch and bound algorithm will produce the optimal solution.

Proof. Assume otherwise. Then the optimal solution belonged to the subtree (possibly of height one) whose lower bound cost was greater than the best cost found so far. By the construction of the lower bound, the optimal solution's cost would be greater than the best cost found so far. Contradiction. □

Algorithm 1 TSP Branch and Bound Algorithm

```
function BnB(G)
    $F() \leftarrow (\emptyset, \infty)$
    ApproxTour, ApproxCost ← Approx(G)
    $q$ ← Priority Queue ordered by Cost
    $v$ ← first node in $G$          ▷ deterministic algorithm
    Put($q$, ($\infty$, [$v$]))
    (Bcost, Btour) ← (ApproxCost, ApproxTour)
    while not Empty($q$) do
        (cost, subtour) ← Get($q$)
        for node $v \in \{V - \bigcup \text{subtour}\}$ do
            subtour ← subtour +$v$
            if $V = \bigcup$ subtour then
                tour ← tour[0]
                cost ← $\sum_{e \in \text{tour}} w(e)$
                if cost<Bcost then
                    (Bcost, Btour) ← (cost, tour)
                end if
            else
                X ← G-subtour
                MST ←MST(X)
                MSTcost ← $\sum_{e \in \text{MST}} w(e)$
                subtourcost ← $\sum_{e \in \text{subtour}} w(e)$
                a← first node in subtour
                b← last node in subtour
                abcost ← $\min_{v \in X} w(a - v) + \min_{v \in X} w(b - v)$
                LB ← MSTcost + subtourcost + abcost
                if cost<Bcost then
                    Put($q$, (cost,subtour))
                end if
            end if
            (cost,path) ← Get($q$)
        end for
    end while
    return tour
end function
```

4.1.3 Data Structures. This algorithm traverses the space of candidate solutions using a priority queue. In the calculation of the lower bound, this algorithm also calculates the minimum spanning tree, which is a subgraph of the input graph. Details of the data structures used in networkx graphs are outlined in the approximation algorithm's data structures section.

4.1.4 Time and Space Complexity. The branch and bound algorithm starts out by creating an initial solution from the approximation algorithm outlined in section 4.2. It then traverses the space of candidate solutions, and in the worst case goes through the $\mathbb{O}(|V|!)$ permutations of cities in a tour. For each candidate solution, this algorithm calculates the cost of the minimum spanning tree and the cost of the partial tour using Kruskal's algorithm and a sum over the edges of the partial tour. This yields a runtime of $\mathbb{O}(|V| + |E|log|E|) + \mathbb{O}(|V|!(|E|log|E| + E)) \in \mathbb{O}(|V|!(|E|log|E|))$.

In addition to the space used to construct the approximate solution that is detailed in section 4.2, this algorithm uses the data structures detailed in the previous section for only one candidate at a time according to when it is called from the priority queue. However, the priority queue at all times contains the children of the frontier of the state space tree that it explores, and for each entry it stores the lower bound cost and the partial tour solution. In the worst case, the number of entries stored in the priority queue grows to $\mathbb{O}(|V|!)$. Thus the priority queue may dominate space complexity with $\mathbb{O}(|V|!|V|)$ dwarfing the space used in the other data structures.

4.1.5 Strengths and Weaknesses. Branch and bound's primary merit is in its ability to eventually reach the optimal solution, making it ideal for small problem sizes. However, its costs grow exponential as it needs to traverse the entire space of candidate solutions, so its not scalable.

4.2 Approximation

4.2.1 Algorithm Description. A minimum spanning tree (MST) heuristic produces a 2-approximation for metric TSP. Our implementation has four main steps. First, Kruskal's Algorithm implemented in networkx obtains the minimum spanning tree[6]. Second, a directed graph constructed from the minimum spanning tree contains its doubled edges. Third, an Eulerian cycle over the directed graph is constructed using an algorithm implemented in the networkx package from Edmonds[1]. Fourth, repeated vertices in the Eulerian cycle are skipped, resulting in a simple cycle that satisfies the definition of a tour. The pseudocode is given in Algorithm 2.

4.2.2 Approximation Guarantee. The tour obtained from the minimum spanning tree construction heuristic provides a 2-approximation.

Proof. By the triangle inequality, the cost of Approx(G) is at most the cost of the double-edged minimum spanning tree. By construction, the cost of the double-edged minimum spanning tree is at most twice the cost of the minimum spanning tree. By relaxation of constraints, the cost of the minimum spanning tree is at most the cost of the minimum tour, which is the cost of the optimal solution. □

4.2.3 Data Structures. The networkx package uses an adjacency list representation for graphs. In detail, it uses a dictionary of dictionaries of dictionaries, where the first dictionary is of adjacency lists for each node, the second is the edge for each neighbor, and the third is the edge data for each edge attribute. The only edge attribute stored is the weight, which is calculated as the euclidean distance between the locations represented by the vertices. The construction of the graph uses

4.2.4 Time and Space Complexity. The algorithm's four steps take time polynomial on the size of the graph. In the first step, Kruskal's algorithm takes a greedy strategy in selecting edges and requires sorting them. In the second step, constructing the doubled directed edges for the minimum spanning tree requires a linear traversal. In the third step, the algorithm used for finding an Eulerian path over the doubled miniminum spanning tree edges required linear time according to its authors [1]. In the fourth step, removing repeated vertices from the Eulerian path requires time linear over the length of the path. This yields time complexity $\mathbb{O}(|E|log|E|) + \mathbb{O}(|V| + |E|) + \mathbb{O}(|V| + |E|) + \mathbb{O}(|V|) \in \mathbb{O}(|V| + |E|log|E|)$. The networkx package uses an adjacency list representation for graphs, and the doubled minimum spanning tree, and Eulerian tour are at most double the original graph's size, giving a space complexity of $\mathbb{O}(|V| + |E|)$.

4.2.5 Strengths and Weaknesses. The approximation guarantee and the time and space complexities justified in the previous sections constitute the chief merits of this algorithm. These guarantees on computing cost and solution quality become observable in our empirical results in our evaluation. In contrast, however, one weakness of this algorithm lies in its relative practical underperformance. For example, although the local search algorithms do not have the same theoretical guarantees, they are able to achieve better results in practice due to their use of stochastic methods.

Algorithm 2 TSP Approximation Algorithm

```
function Approx(G)
    MST ← Kruskal's Algorithm[ref](G)
    DG ← directed, symmetric G
    Cycle ← Eulerian Circuit[ref](DG)
    Visited ←
    Tour ← []
    for vertex $v \in$ Cycle do
        if $v \notin$ Visited then
            Tour ← Tour + [$v$]
            Visited ← Visited $\cup$ $v$
        end if
    end for
end function
Cost = 0
for edge $e \in$ Tour do Cost ← Cost + dist($e$)
end for
return Tour, Cost
```

4.3 Simulated Annealing

Simulated Annealing is a probabilistic technique used for optimization problems. It's adopted from Physics where the temperature profile of a hot metal that is being cooled flattens over time. The underling equation behind Simulated Annealing is the probability of transitioning between two energy states:

$$P(e_c, e_n, T) = \exp\left(\frac{-\Delta E}{T}\right)$$

where
e_c = energy of the current configuration
e_n = energy of the next configuration
T = current temperature
ΔE = Difference in the energy states

The crux of simulated annealing is how we go about generating neighboring candidates so that we can move in the path of a better solution. In this work, we generate the next neighbor by randomly switching 2 nodes along the sequence and we check if the energy is lesser. If it is, then $\Delta E > 0$ and we pick the generated neighbour as the potential candidate. However, the important part is that even if the neighbour has higher energy, we assign it a small non zero probability of being picked. This is similar to the $\epsilon-$ greedy algorithm in the bandit reinforcement learning problem where we seek to explore occasionally in the hope of finding a lower cost trajectory. In the context of Simulated Annealing, this random higher energy solutions allows us to escape local minima and cover the entire optimization landscape.

When annealing begins, the temperature is high and we tend to bounce off a lot in the cost function landscape. Over time, the temperature is cooled and the acceptance criteria becomes strict, giving a method to this madness behavior. This is similar to stochastic gradient descent (SGD) where the random behavior reduces as we get closer to the optimal value. The search space becomes narrower and narrower taking us closer to the true optimal value of the cost function.

Cooling Schedule

This is the most important aspect of SA algorithms. Cooling the temperature very fast makes the algorithm freeze while a slow schedule might take forever for convergence. A right balance is thus required. Following schedules are common in practice:

$$T = T_0 - \alpha i \quad \text{Linear Multiplicative}$$

$$T = T_0 \alpha^i \quad \text{Exponential Multiplicative}$$

$$T = \frac{T_0}{1 + \alpha \log(1+i)} \quad \text{Log scale Multiplicative}$$

$$T = \frac{T_0}{1 + \alpha i^2} \quad \text{Quadratic Multiplicative}$$

where,
T_0 is is the starting temperature, i is the iteration number, and α the most important hyper parameter is the rate of cooling.

We tried quadratic and log scale cooling schedule initially for a smooth transitions but it was bit of a overkill. Fast cooling schedule like the exponential multiplicative one seemed just fine that resulted in closer to the optimal solution in a much shorted span of time. To gave a generalized hyper parameter setting, T_0 is set to be the square root of the number of node instances. This makes sense because if the size of the graph is large there are many possible candidates and to narrow done to the best one we need to explore a lot meaning start with a high temperature. α was set to be 0.999 across all the runs.

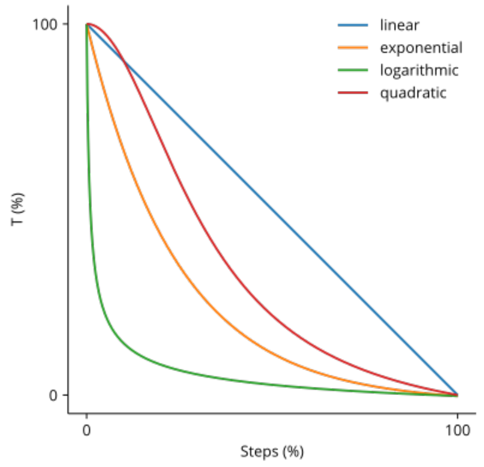


Fig 1:
Cooling schedules in simulated Annealing

Algorithm 3 TSP Local Search Algorithm - SA

```
1:  function SIMULATED ANNEALING(ProblemSize)
2:      $S_{current} \leftarrow$ SCREATESOLUTION(ProblemSize)
3:      $S_{best} \leftarrow S_{current}$
4:      for $i$ in range(iter) do
5:          $S_i \leftarrow$ CREATENEIGHBORSOLUTION($S_{current}$)
6:          $temp_{current} \leftarrow$ CALCULATETEMP($i, temp_{max}$)
7:          if $cost(S_i) \leq cost(S_{current})$ then
8:              $S_{current} \leftarrow S_i$
9:              if $cost(S_i) \leq cost(S_{best})$ then
10:                 $S_{best} \leftarrow S_i$
11:             end if
12:         else if $exp\frac{cost(S_{current}) - cost(S_i)}{temp_{current}}$ >Rand() then
13:             $S_{current} \leftarrow S_i$
14:         end if
15:     end for
16:     return $S_{best}$
17: end function
```

4.4 Complexities

SA goes through $O(n^2)$ to look for pairs for n points in the graph. It takes $O(n)$ to go through the search space inside the while loop. The rejection for temperature costs $O(1)$ while acceptance of a temperature chance is the average path reversal cost of $O(n!)$. Therefore, the overall time complexity is $O(n \cdot n!)$. The space complexity is $O(n^2)$ since the data structure is a graph dictionary.

4.5 Strength and Weaknesses

Advantages of the simulated annealing algorithms are that its easier to implement even for complex problems, can handle arbitrary cost functions and generally gives a good solution. However, when the cost function landscape is smooth, simulated annealing might be bit of an overkill and also can't tell if we have reached the global minima (some method like branch and bound might be needed for this).

4.6 Genetic Algorithm

The genetic algorithm comes under the tree of local search meta-heuristics. It is heavily inspired by evolutionary principles, in specific - the process of natural selection. Intuitively, the stages of :

- Survival of the fittest
- Crossover Mating
- Random Mutations

The core principles/terminologies are first discussed as follows. Later, we discuss the adaptation of this meta-heuristic to our TSP scenario.

Introduction. The GA procedure starts off with choosing a random population of a genetic sequence (DNA). Each genetic sequence is composed of single units known as genes, and each gene represents a specific parameter of the solution. The neighborhood for the local search is composed of these DNA populations and their succeeding generations. Following are the core functions of the process.

Selection. It is intuitive that the fittest DNA is one that needs to survive and get passed on to the next generation. The selection process follows this procedure in the hopes of retaining parameters that optimize fitness. The fitness of each DNA is thus calculated and the resulting total fitness is to be maximized. Choosing DNA is usually done by using selecting probabilistically, with maximum fitness being given the maximum weight.

Elitism. This concept refers to keeping certain amounts of 'fit' DNA as is from one generation to the next. This ensures that the fit DNAs are preserved and allows us to lower bound on the fittest DNA of the previous generation.

Mating. The concept of crossover mating allows children to inherit 'features' or parameters from it's parents. This is repeated till the whole next generation is created.

Mutation. For mutation, a gene is randomly replaced with another generated gene.The mutation allows variation in the population, and hence allows the GA to stay away from sticking to local optimums.

Termination. There is no guarantee that the best solution is eventually found due to the problem being fundamentally reliant on random selection. Hence, the termination is done after iterating through a maximum number of generations. Finally, in the context of TSP, we see as follows -

- **Gene** : A gene is a location in a city (represented by (x, y) coordinates).
- **DNA** : A solution to the TSP satisfying TSP constraints (each location can be visited exactly and it ends up in the same node it started).
- **Fitness** : Reciprocal of tour lengths.
- **Population** : A collection of individuals (A set of routes).
- **Mating pool** : A collection of parents that are used to create the subsequent population.
- **Mutation** : A method to introduce variation to our population by switching nodes at random.
- **Elitism** : A way to carry the best individuals to the next generation - percentage of the population with the fittest individuals

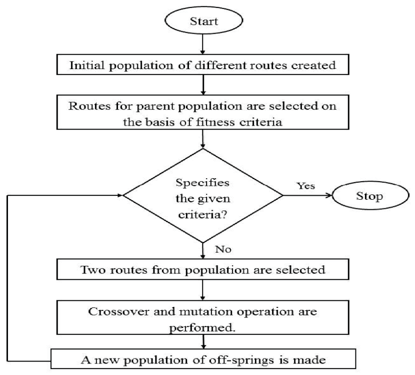
Algorithm 4 TSP Local Search Algorithm - GA

```
1: function geneticAlgorithm
2:     population ← InitialPopulation((population-size, nodeList))
3:     for $i$ in range(generations) do
4:         popRanked ← rankRoutes(currentGen)
5:         selectionResults ← selection(popRanked, elite-size)
6:         matingPool ← matingPool(currentGen, selectionResults)
7:         children ← breedPopulation(matingpool, eliteSize)
8:         nextGeneration =← mutatePopulation(children, mutationRate)
9:         population ← nextGeneration
10:    end for
11:    $finalDist$ ← int(1 / rankRoutes(population)[0][1])
12:    bestRouteIndex = rankRoutes(population)[0][0]
13:    bestRoute ← population[bestRouteIndex]
14:    return bestRoute, $finalDist$
15: end function
```

Time Complexity. Let n_p be the population size, n_g be the number of generations and N be the number of nodes in a selected city. Following are the step-specific complexities.

- **Initialize** : $O(N \cdot n_p)$
- **Select** : $O(N \cdot n_p) + O(n_p \cdot \log(n_p))$
- **Mating** : $O(N \cdot n_p)$
- **Mutation** : $O(N \cdot n_p)$

Figure 1. Flowchart of the Genetic Algorithm methodology


Start
Initial population of different routes created
Routes for parent population are selected on the basis of fitness criteria
Specifies the given criteria?
Yes
Stop
No
Two routes from population are selected
Crossover and mutation operation are performed.
A new population of off-springs is made

- **Total** : $O(N \cdot n_p \cdot n_g)$

Total time complexity : We see that the problem is in $O(N)$. Although it varies linearly with number of nodes, n_p and n_g are in fact quite large. Due to this, N has less contribution to the actual complexity.

Space Complexity. **Total space complexity** : $O(N \cdot n_p)$

Hyperparameters. As discussed previously, the hyperparameters for this problem are mentioned as seen below. They are empirically chosen to match computational capabilities of the subsystem.

- **popSize** : This corresponds to the amount of DNA available in pool for mating. A large size allows us to introduce more variation and ultimately produce a better solution. But, using a larger population would require more computational time as well. We use `popSize` = 100.
- **eliteSize** : The number of fittest DNAs from the population that get carried over to the next generation. If we take a high number for this, there will be a fitter next generation, but it does not allow us to take advantage of variation. In turn, it makes the problem more likely to be stuck at a local optimum. Also, if the eliteSize is 0, then there is no guaranteed lower bound to the solution. We use `eliteSize` = 20.
- **mutationRate** : This is the probability of mutation for a gene. Allowing this probability to be high makes the children inherit features at random instead of similar to the parents. This is a bad decision as the solution may never be optimal due to randomness. We use `mutationRate` = 0.01.
- **generations** : We use `generations` = 100.

Strength and Weaknesses. We implemented most of the code with the help of lists as the data-structures. They are known to be easy to implement with python but also at the price of computational power. It is well known that GA usually reaches close to the neighborhood of the optimum fairly quickly. But, it takes a long time to reach the actual optimum. This is due to the stochastic nature of the algorithm. A good use of GA would be to reduce the solution space. After this is done, it is better for another algorithm to take over if we want to min-max runtime-optimality. The interpretability of GA makes it elegant and intuitive and it is akin to the real life process of evolution.

5 Empirical Evaluation

We tested the above mentioned four algorithms with the metrics being quality of solution : length of path and relative error, and the running time of the algorithm. We have mentioned tables for all four, and solution quality distributions and boxplots for the local search heuristics. First, we look at our system specifics.

- **OS :** Linux - Ubuntu 20.04.2 LTS
- **Memory :** 8GB RAM
- **Processor :** Intel core i7-9750H CPU @ 2.60GHz × 12
- **Python :** 3.8

5.1 Branch and Bound

Dataset	Time(s)	Solution Quality	Relative Error
Atlanta	32.30	2003763	0.0000
Berlin	1.67	8091	0.0728
Boston	400.31	955440	0.0693
Champaign	490.60	53493	0.0161
Cincinnati	0.03	277952	0.0000
Denver	19.78	109235	0.0877
NYC	50.89	1774194	0.1409
Philadelphia	257.50	1437663	0.0298
Roanoke	42.06	780178	0.1903
SanFrancisco	2.67	896718	0.1068
Toronto	239.04	1220114	0.0374
UKansasState	0.01	62962	0.0000
UMissouri	296.47	156572	0.1798

5.2 Approximation Heuristic - MST

Dataset	Time(s)	Solution Quality	Relative Error
Atlanta	0.00	2415132	0.2053
Berlin	0.01	10303	0.3661
Boston	0.00	1094649	0.2251
Champaign	0.01	61508	0.1684
Cincinnati	0.00	315452	0.1349
Denver	0.02	126189	0.2565
NYC	0.01	1884293	0.2117
Philadelphia	0.00	1722655	0.2340
Roanoke	0.10	797872	0.2173
SanFrancisco	0.02	1102577	0.3609
Toronto	0.03	1686472	0.4339
UKansasState	0.00	70143	0.1141
UMissouri	0.02	153757	0.1586

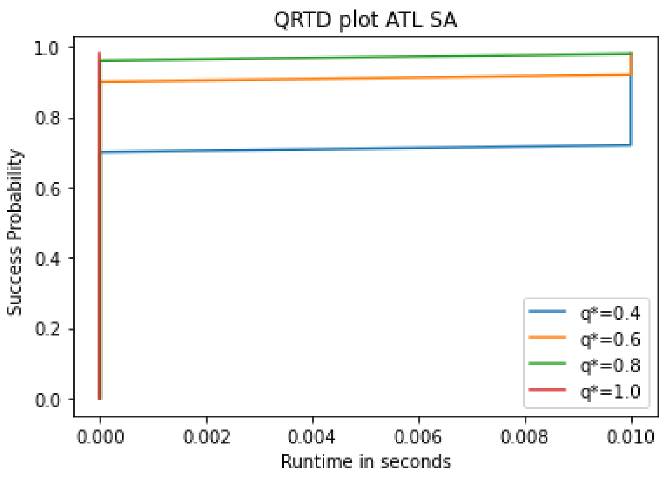


Figure 2. QTRD plot for Atlanta - SA

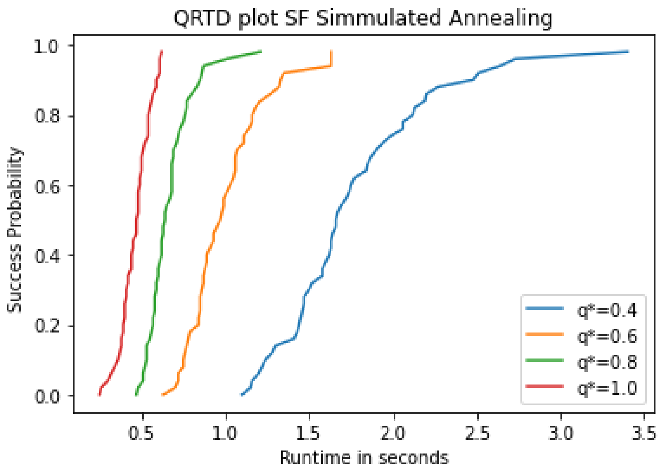


Figure 3. QTRD plot for San Francisco - SA

5.3 Local Search - Simulated Annealing

Dataset	Time(s)	Solution Quality	Relative Error
Atlanta	0.10	2048953	0.02
Berlin	1.85	8293	0.09
Boston	0.58	937695	0.04
Champaign	1.79	55164	0.04
Cincinnati	0.04	280239	0.09
Denver	4.28	109891	0.09
NYC	2.97	1665475	0.07
Philadelphia	0.14	1414215	0.01
Roanoke	12.81	1045020	0.59
SanFrancisco	5.31	952265	0.17
Toronto	5.88	1266941	0.16
UKansasState	0.03	62962	0.00
UMissouri	5.53	151857	0.14

The QTRD plots for Simulated Annealing are seen in figures 2 and 3. Box plots are seen in figures 4 and 5. SQD plots are seen in figures 7 and 6.

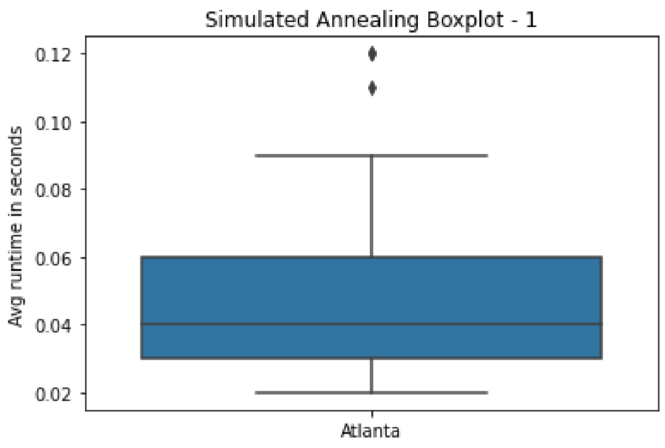


Figure 4. Box plot for Atlanta - SA

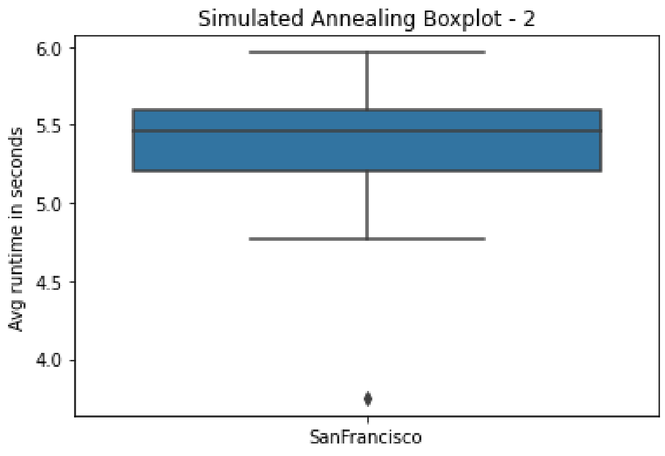


Figure 5. Box plot for San Francisco - SA

5.4 Local Search - Genetic Algorithm

Dataset	Time(s)	Solution Quality	Relative Error
Atlanta	5.76	2106137	0.05
Berlin	8.10	14731	0.95
Boston	7.55	13289013	0.48
Champaign	8.24	109765	1.08
Cincinnati	1.41	278181	0.00
Denver	9.90	335853	2.34
NYC	9.12	3893370	1.51
Philadelphia	6.66	1606507	0.15
Roanoke	18.47	5051430	6.71
SanFrancisco	11.01	3788307	3.67
Toronto	11.42	6304090	4.35
UKansasState	1.49	62962	0.00
UMissouri	11.51	493231	2.71

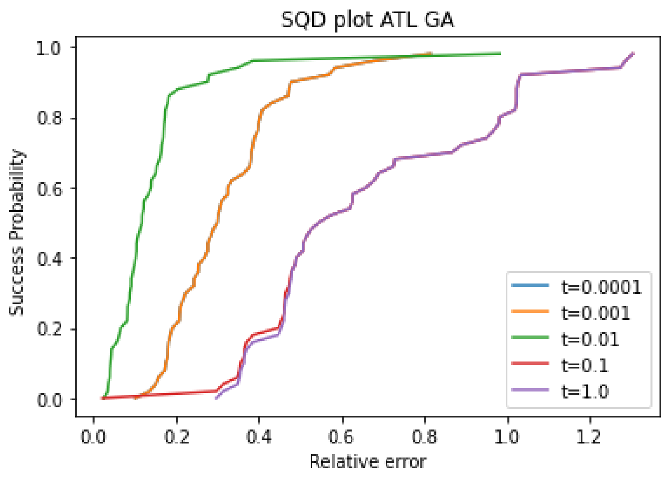


Figure 6. SQD plot for Atlanta - SA

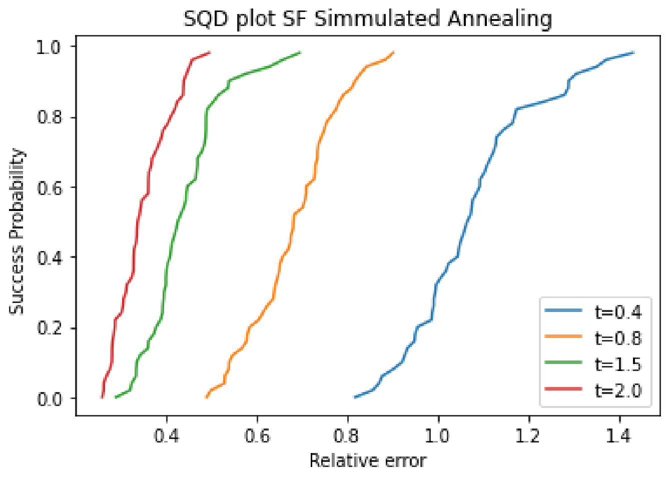


Figure 7. SQD plot for San Francisco - SA

The QTRD plots for Genetic Algorithm are seen in figures 8 and 9. Box plots are seen in figures 10 and 11. SQD plots are seen in figures 13 and 12.

6 Discussion

As expected, the performance of the approximation - MST solution is extremely good. The implementation is easy and the run-time is extremely less as well. Comparing this to the Branch and Bound, we notice that in this case the run-times are good for smaller datasets and for the bigger ones, it is significantly and disproportionately increased. The time complexity of this implementation is in the order of $O(n!)$ and explaining this sort of increase. Also, the branch and bound solution gives an exact solution in the case of smaller datasets which makes sense because branch and bound always gives an optimal solution if we wait for long enough. Since our branch and bound implementation starts with an

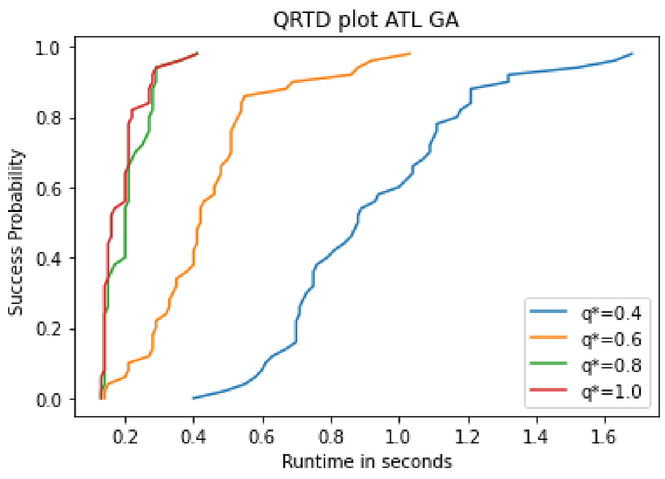


Figure 8. QTRD plot for Atlanta - GA

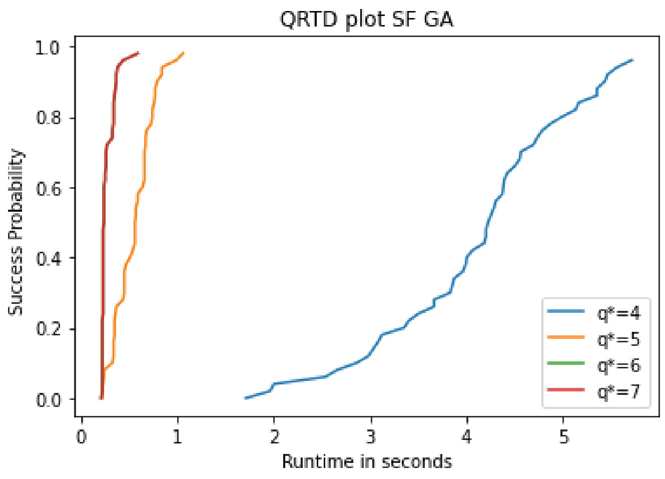


Figure 9. QTRD plot for San Francisco - GA

initial solution that uses the minimum spanning tree construction heuristic, a 2-approximation, its results fit our expectation of a relative error of at most 100%. Next we look at local search algorithms. We used two algorithms based on natural sciences, the Simulated annealing algorithm - with the concept originating from the field of metallurgy and the Genetic algorithm, from the field of evolutionary biology. Our results show that in the simulated annealing algorithm, an increase in input size adversely affects our output time. It's strength lies in the ease of implementation and it's easy intuition. For a dataset with a large neighborhood, SA performs poorly as each iteration includes only one additional node. In the case of Genetic Algorithms, our results were sadly quite disappointing but as expected from literature. From the QTRD plots we see that we can reach an acceptable level of optimality quickly, but going closer to optimal solutions take a considerable amount of time. For these reasons,

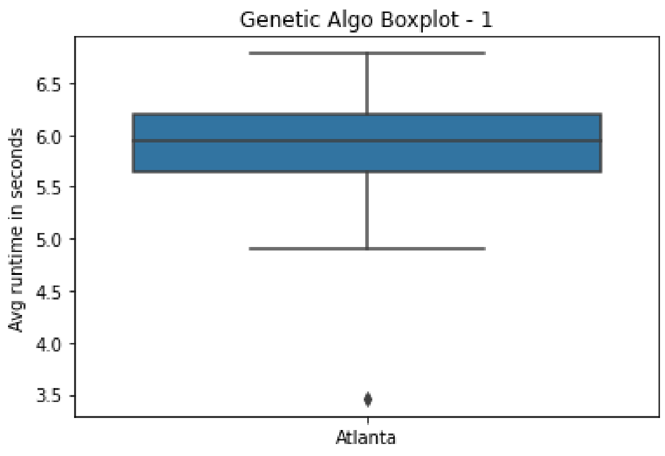


Figure 10. Box plot for Atlanta - GA

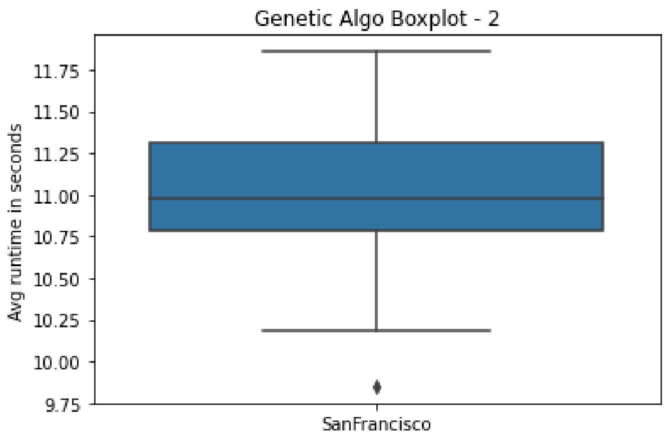


Figure 11. Box plot for San Francisco - GA

it is generally expected to use GA only to get to the approximate neighborhood of the solution and then use a different algorithm to reach quick and stable convergence. Marginal improvements take a lot of time and this is attributed to the random nature of the algorithm.

7 Conclusion

The metric traveling salesman problem has great utility but great difficulty, and the algorithms presented in this paper offer a way to reach reasonable results in practice. Each has strengths and weaknesses suited to different scenarios. For instance, the branch and bound algorithms succeed in providing perfect solutions for the smallest datasets. This is evident in our branch and bound algorithm's performance on the cities with only ten locations. On the other hand, approximation algorithms are excellent for quickly providing solutions with guaranteed small relative errors. Our simple minimum spanning tree construction heuristic found solutions for every city within less than a second that had relative errors

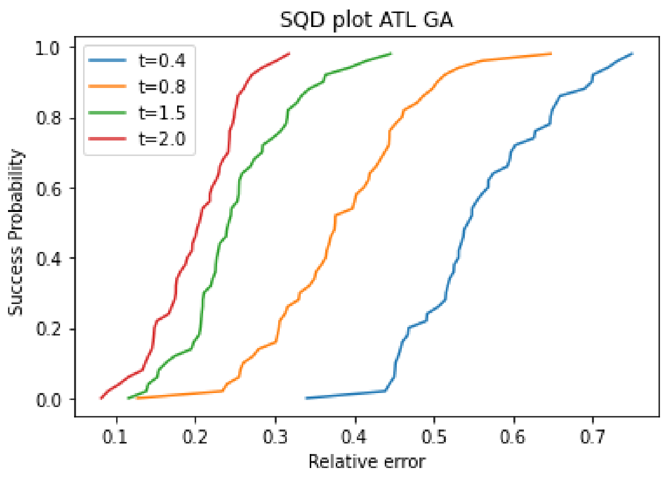


Figure 12. Box plot for Atlanta - GA

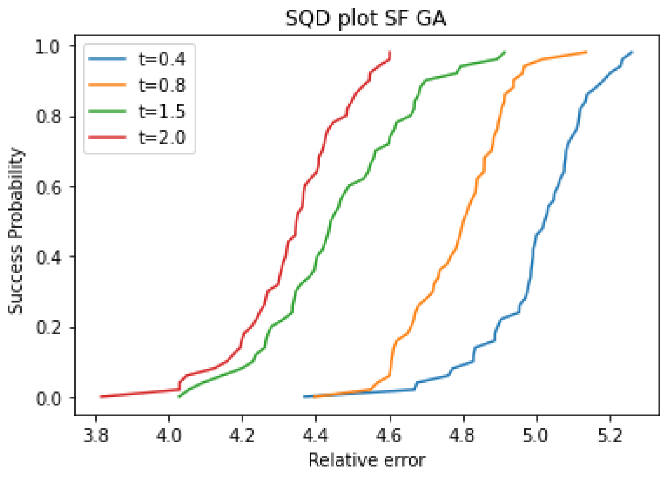


Figure 13. Box plot for San Francisco - GA

well-within the expected theoretical bound of 2. Local search algorithms perform extremely well with the caveat of the difficulty in hyperparameter optimization. For example, the genetic algorithm often got close to the correct solution very quickly but then had difficulty and took great time in converging to termination. The simulated annealing algorithm was the fast of all four algorithms we investigated when its cooling schedule was correctly selected, but determining the schedule proved a challenging task. In this way, one should consider interest in solution quality, time, and hyperparameter optimization before making a selection from the range of algorithms available to tackle metric TSP.

References

[1] Jack Edmonds and Ellis Johnson. 1973. Matching, Euler Tours and the Chinese Postman. *Mathematical Programming* 5 (12 1973), 88–124. https://doi.org/10.1007/BF01580113

[2] David B Fogel. 1993. Applying evolutionary programming to selected traveling salesman problems. *Cybernetics and systems* 24, 1 (1993), 27–36.

[3] B. Golden, L. Bodin, T. Doyle, and W. Stewart. 1980. Approximate Traveling Salesman Algorithms. *Operations Research* 28, 3 (1980), 694–711. http://www.jstor.org/stable/170036

[4] Michael Held and Richard M. Karp. 1970. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research* 18, 6 (1970), 1138–1162. http://www.jstor.org/stable/169411

[5] John Henry Holland et al. 1992. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press.

[6] Joseph B. Kruskal. 1956. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. Amer. Math. Soc.* 7, 1 (1956), 48–50. http://www.jstor.org/stable/2033241

[7] A. H. Land and A. G. Doig. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28, 3 (1960), 497–520. http://www.jstor.org/stable/1910129

[8] Martin Pincus. 1970. Letter to the editor—a Monte Carlo method for the approximate solution of certain types of constrained optimization problems. *Operations research* 18, 6 (1970), 1225–1228.

[9] René van Bevern and Viktoriia A. Slugina. 2020. A historical note on the 3/2-approximation algorithm for the metric traveling salesman problem. *Historia Mathematica* 53 (2020), 118–127. https://doi.org/10.1016/j.hm.2020.04.003

[10] Mike Worboys. 1986. The travelling salesman problem (A guided tour of combinatorial optimisation), edited by E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. Pp 465. £39·95. 1985. ISBN 0-471-90413-9 (Wiley). *The Mathematical Gazette* 70, 454 (1986), 327–328. https://doi.org/10.2307/3616224